Velocity Template User Guide

Office of Operational Services (OOS)
04/24/2008

This document is only valid for TO-8 Release of AWIPSII.
It will have to be updated as new information comes available and with subsequent releases. It should be considered for informational purposes only.

About this Guide

The Velocity User Guide is intended to help template editors get acquainted with Velocity and the syntax of its scripting language, the Velocity Template Language (VTL). Many of the examples in this guide deal with using Velocity to embed dynamic content into text Formatters. This document was modified from the original document located at http://velocity.apache.org/engine/devel/user-guide.html

What is Velocity?

Velocity is a Java-based template engine. It permits the templates to reference methods defined in Java code. Template designers may have to understand the methods, and objects in the java code since they are used throughout the templates. Velocity separates Java code from the warning products, making the templates more maintainable over the long run.
The exact details of the *foreach* statement will be described in greater depth shortly; what's important is the impact this short script can have on your web site. When a customer with a penchant for Bright Red Mud logs in, and Bright Red Mud is on sale, that is what this customer will see, prominently displayed. If another customer with a long history of Terracotta Mud purchases logs in, the notice of a Terracotta Mud sale will be front and center. The flexibility of Velocity is enormous and limited only by your creativity.
Documented in the VTL Reference are the many other Velocity elements, which collectively give you the power and flexibility you need to make your web site a web *presence*. As you get more familiar with these elements, you will begin to unleash the power of Velocity.

Velocity Template Language (VTL): An Introduction

VTL uses *references* to embed dynamic content in a text Product, and a variable is one type of reference. Variables are one type of reference that can refer to something defined in the Java code, or it can get its value from a VTL *statement* in the template itself. Here is an example of a VTL statement that could be embedded in an template document:

```
#set( $a = "National Weather Service" )
```

This VTL statement, like all VTL statements, begins with the # character and contains a directive: *set*. When the warning product is generated, the Velocity Templating Engine will search through template to find all # characters, then determine which mark the beginning of VTL statements, and which of the # characters that have nothing to do with VTL.

The # character is followed by a directive, *set*. The *set* directive uses an expression (enclosed in brackets) -- an equation that assigns a *value* to a *variable*. The variable is listed on the left hand side and its value on the right hand side; the two are separated by an = character.

In the example above, the variable is *$a* and the value is *National Weather Service*. This variable, like all references, begins with the *$* character. String values are always enclosed in quotes, either single or double quotes. Single quotes will ensure that the quoted value will be assigned to the reference as is. Double quotes allow you to use velocity references and directives to interpolate, such as "National Weather Service $office", where the *$office* will be replaced by the current value before that string literal is assigned to the left hand side of the =

The following rule of thumb may be useful to better understand how Velocity works: **References begin with *$* and are used to get something. Directives begin with *#* and are used to do something.**

In the example above, *#set* is used to assign a value to a variable. The variable, *$a*, can then be used in the template to output "Velocity".

Templates!

Once a value has been assigned to a variable, you can reference the variable anywhere in your template. In the following example, a value is assigned to *$office* and later referenced.

```
THIS IS A TEST
TEST TEST TEST
#set( $office = "MANHATTAN, KANSAS" )
THE NATIONAL WEATHER SERVICE IN $office HAS DONE NOTHING.
```

The result is a text product that prints:

```
THIS IS A TEST
TEST TEST TEST
THE NATIONAL WEATHER SERVICE IN MANHATTAN, KANSAS HAS DONE
NOTHING.
```

To make statements containing VTL directives more readable, we encourage you to start each VTL statement on a new line, although you are not required to do so. The *set* directive will be revisited in greater detail later on.

Comments

Comments allows descriptive text to be included that is not placed into the output of the template engine. Comments are a useful way of reminding yourself and explaining to others what your VTL statements are doing, or any other purpose you find useful. Below is an example of a comment in VTL.

This is a single line comment.
A single line comment begins with ## and finishes at the end of the line. If you're going to write a few lines of commentary, there's no need to have numerous single line comments. Multi-line comments, which begin with #* and end with *#, are available to handle this scenario.

This is text that is outside the multi-line comment.
It will be inserted into the text product.

#*
 Thus begins a multi-line comment. The text product will not contain this text because the Velocity Templating Engine will
 ignore it.
*#
Here is text outside the multi-line comment; it is visible.

Here are a few examples to clarify how single line and multi-line comments work:

This text is visible. ## This text is not.

This text is visible.

This text is visible.
#* This text, as part of a multi-line comment, is not visible. This text is not visible; it is also part of the multi-line comment. This text still not visible. *#

This text is outside the comment, so it is visible.

This text is not visible.

There is a third type of comment, the VTL comment block, which may be used to store such information as the document author and versioning information:
#**
This is a VTL comment block and may be used to store such information as the document author and versioning information:
@author
@version 5
*#

References

There are three types of references in the VTL: variables, properties and methods. As a template editor, you will have to understand the specific names of references in the JAVA code so you can use them correctly in your templates. *See the separate document "Reference Reference" to be completed at a later date.*

Everything coming to and from a reference is treated as a String object. If there is an object that represents *$wfo* (such as an Integer object), then Velocity will call its `.toString()` method to resolve the object into a String.

Variables

The shorthand notation of a variable consists of a leading "$" character followed by a VTL *Identifier*. A VTL Identifier must start with an alphabetic character (a .. z or A .. Z). The rest of the characters are limited to the following types of characters:

alphabetic (a .. z, A .. Z)
numeric (0 .. 9)
hyphen ("-")
underscore ("_")

Here are some examples of valid variable references in the VTL:

${officeLong}
${testMessage}
${stormType}
${pathCast}

When VTL references a variable, such as *$officeLong*, the variable can get its value from either a *set* directive in the template, or from the Java code. For example, if the Java variable *$officeLong* has the value *SST* at the time the template is requested, SST replaces all instances of *$officeLong* in the text product. Alternatively, if I include the statement:

#set($officeLong = "OST")

The output will be the same for all instances of *$officeLong* that follow this directive.

Properties

The second flavor of VTL references are properties, and properties have a distinctive format. The shorthand notation consists of a leading *$* character followed a VTL Identifier, followed by a dot character (".") and another VTL Identifier. These are examples of valid property references in the VTL:

${closest.roundedDistance}
${closest.roundedAzimuth}

Take the first example, *${closest.roundedDistance}*. It can have two meanings. It can mean, Look in the hashtable identified as *closest* and return the value associated with the key *roundedDistance*. But *${closest.roundedDistance}* can also be referring to a method (references that refer to methods will be discussed in the next section);

${closest.roundedDistance}* could be an abbreviated way of writing *${closest.roundedDistance()}. When your template is processed, Velocity will determine

which of these two possibilities makes sense, and then return the appropriate value.

Methods

A method is defined in the Java code and is capable of doing something useful, like running a calculation or arriving at a decision. Methods are references that consist of a leading "$" character followed a VTL Identifier, followed by a VTL *Method Body*. A VTL Method Body consists of a VTL Identifier followed by an left parenthesis character ("("), followed by an optional parameter list, followed by right parenthesis character (")"). These are examples of valid method references in the VTL:

${dateUtil.format(${now}, ${timeFormat.ddhhmm})}
${mathUtil.round(${movementDirection})}

These two examples -- *${dateUtil.format()}* and *${mathUtil.round()}* -- may look similar to those used in the Properties section above, *${closest.roundedDistance}* and *${closest.roundedAzimuth}* . If you guessed that these examples must be related some in some fashion, you are correct!

VTL Properties can be used as a shorthand notation for VTL Methods. The Property *$closest.roundedDistance* has the exact same effect as using the Method *$closest.getRoundedDistance()*. It is generally preferable to use a Property when available. The main difference between Properties and Methods is that you can specify a parameter list to a Method.

Property Lookup Rules

As was mentioned earlier, properties often refer to methods of the parent object. Velocity is quite clever when figuring out which method corresponds to a requested property. It tries out different alternatives based on several established naming conventions. The exact lookup sequence depends on whether or not the property name starts with an upper-case letter. For lower-case names, such as *$customer.address*, the sequence is

getaddress()
getAddress()
get("address")
isAddress()

For upper-case property names like *$customer.Address*, it is slightly different:

getAddress()
getaddress()
get("Address")
isAddress()

Formal Reference Notation

Shorthand notation for references was used for the examples listed above, but there is also a formal notation for references, which is demonstrated below:

${mudSlinger}
${customer.Address}
${purchase.getTotal()}

In almost all cases you will use the shorthand notation for references, but in some cases the formal notation is required for correct processing.
Suppose you were constructing a sentence on the fly where *$vice* was to be used as the base word in the noun of a sentence. The goal is to allow someone to choose the base word and produce one of the two following results: "Jack is a pyromaniac." or "Jack is a kleptomaniac.". Using the shorthand notation would be inadequate for this task. Consider the following example:

Jack is a $vicemaniac.

There is ambiguity here, and Velocity assumes that *$vicemaniac*, not *$vice*, is the Identifier that you mean to use. Finding no value for *$vicemaniac*, it will return *$vicemaniac*. Using formal notation can resolve this problem.
Jack is a ${vice}maniac.

Now Velocity knows that *$vice*, not *$vicemaniac*, is the reference. Formal notation is often useful when references are directly adjacent to text in a template.

Quiet Reference Notation

When Velocity encounters an undefined reference, its normal behavior is to output the image of the reference. For example, suppose the following reference appears as part of a VTL template.

Email Address ="$email"

When the form initially loads, the variable reference *$email* has no value, but you prefer a blank text field to one with a value of "$email". Using the quiet reference notation circumvents Velocity's normal behavior; instead of using *$email* in the VTL you would use *$!email*. So the above example would look like the following:

Email Address = $!email

Now when the form is initially loaded and *$email* still has no value, an empty string will be output instead of "$email".

Formal and quiet reference notation can be used together, as demonstrated below.
Emal Address=$!{email}

Getting literal

VTL uses special characters, such as *$* and *#*, to do its work, so some added care should be taken where using these characters in your templates. This section deals with escaping the *$* character.

Escaping Valid VTL References

Cases may arise where there is the potential for Velocity to get confused. *Escaping* special characters is the best way to handle VTL's special characters in your templates, and this can be done using the backslash (\ ) character.

#set($email = "my.mail@noaa.gov")

$email

If Velocity encounters a reference in your VTL template to *$email*, it will search the Context for a corresponding value. Here the output will be *my.mail@noaa.gov*, because *$email* is defined. If *$email* is not defined, the output will be *$email*.
Suppose that *$email* is defined (for example, if it has the value *my.mail@noaa.gov*), and that you want to output *$email*. There are a few ways of doing this, but the simplest is to use the escape character.

The following line defines $email in this template:
#set($email = "my.mail@noaa.gov")

$email
\$email
\\$email
\\\$email

renders as

my.mail@noaa.gov
$email
\my.mail@noaa.gov
\$email

Note that the \ character bind to the *$* from the left. The bind-from-left rule causes *\\\$email* to render as *\\$email*. Compare these examples to those in which *$email* is not defined.

```
$email
\$email
\\$email
\\\$email
```

renders as

```
$email
\$email
\\$email
\\\$email
```

Notice Velocity handles references that are defined differently from those that have not been defined. Here is a set directive that gives *$wfo* the value *gibbous*.

```
#set( $wfo = "MAN" )
$myWfo = $wfo
```

The output will be: ***$myWfo = MAN*** -- where *$myWfo* is output as a literal because it is NOT defined and *MAN* is output in place of *$wfo*.

It is also possible to escape VTL directives; this is described in more detail in the Directives section.

Case Substitution

Now that you are familiar with references, you can begin to apply them effectively in your templates. Velocity references take advantage of some Java principles that template designers will find easy to use. For example:

```
$wfo

$wfo.getRfc()
## is the same as
$wfo.Rfc

$data.setUser("jon")
## is the same as
#set( $data.User = "jon" )

$data.getRequest().getServerName()
## is the same as
```

$data.Request.ServerName
is the same as
${data.Request.ServerName}

These examples illustrate alternative uses for the same references. Velocity takes advantage of Java's introspection and bean features to resolve the reference names to both objects in the Context as well as the objects methods. It is possible to embed and evaluate references almost anywhere in your template.

Velocity, which is modelled on the Bean specifications defined by Sun Microsystems, is case sensitive; however, its developers have strove to catch and correct user errors wherever possible. When the method *getWfo()* is referred to in a template by `$rfc.wfo`, Velocity will first try `$getwfo`. If this fails, it will then try `$getWfo`. Similarly, when a template refers to `$rfc.Wfo`, Velocity will try *$getWfo()* first and then try *getwfo()*.

Note: *References to instance variables in a template are not resolved.* Only references to the attribute equivalents of JavaBean getter/setter methods are resolved (i.e. `$wfo.Name` does resolve to the class Wfo's `getName()` instance method, but not to a public `Name` instance variable of Wfo).

Directives

References allow template editors to generate dynamic content for text products, while *directives* -- easy to use script elements that can be used to creatively manipulate the output of Java code -- permit template editors to truly take charge of the appearance and content of the text product output.

Directives always begin with a `#`. Like references, the name of the directive may be bracketed by a `{` and a `}` symbol. This is useful with directives that are immediately followed by text. For example the following produces an error:

#if($a==1)true enough#elseno way!#end

In such a case, use the brackets to separate `#else` from the rest of the line.

#if($a==1)true enough#{else}no way!#end
#set

The *#set* directive is used for setting the value of a reference. A value can be assigned to either a variable reference or a property reference, and this occurs in brackets, as demonstrated:

#set($myString = "TAKE COVER IMMEDIATELY")
#set($outText.callToAction = $myString)

The left hand side (LHS) of the assignment must be a variable reference or a property reference. The right hand side (RHS) can be one of the following types:

Variable reference
String literal
Property reference
Method reference
Number literal
ArrayList
Map

These examples demonstrate each of the aforementioned types:

```
#set( $office = $officeLong ) ## variable reference
#set( $office.Phone = "888-555-1212" ) ## string literal
#set( $office.Address = $local.Address ) ## property reference
#set( $office.AreaCode = $local.AreaCode($code)) ## method reference
#set( $office.Zip = 123 ) ##number literal
#set( $office.List = ["ITO", $my, "ESA"] ) ## ArrayList
#set( $hail.Map = {"penny" : "1/2 inch","quarter" : "1 inch"}) ## Map
```

NOTE: For the ArrayList example the elements defined with the [..] operator are accessible using the methods defined in the ArrayList class. So, for example, you could access the first element above using $local.Address.get(0).

Similarly, for the Map example, the elements defined within the { } operator are accessible using the methods defined in the Map class. So, for example, you could access the first element above using $hail.Map.get("penny") to return a String '1/2 inch', or even $hail.Map.penny to return the same value.

The RHS can also be a simple arithmetic expression:

```
#set( $value = $rain + 1 )
#set( $value = $snow - 1 )
#set( $value = $rain * $snow )
#set( $value = $snow / $rain )
```

If the RHS is a property or method reference that evaluates to *null*, it will **not** be assigned to the LHS. Depending on how Velocity is configured, it is usually not possible to remove an existing reference from the context via this mechanism. (Note that this can be permitted by changing one of the Velocity configuration properties). This can be confusing for newcomers to Velocity. For example:

```
#set( $result = $query.criteria("spotter") )

The result of the first query is $result
```

```
#set( $result = $query.criteria("address") )

The result of the second query is $result
```

If *$query.criteria("spotter")* returns the string "bill", and *$query.criteria("address")* returns *null*, the above VTL will render as the following:

```
The result of the first query is bill

The result of the second query is bill
```

This tends to confuse newcomers who construct *#foreach* loops that attempt to *#set* a reference via a property or method reference, then immediately test that reference with an *#if* directive. For example:

```
#set( $criteria = ["spotter", "address"] )

#foreach( $criterion in $criteria )

   #set( $result = $query.criteria($criterion) )

   #if( $result )
      Query was successful
   #end

#end
```

In the above example, it would not be wise to rely on the evaluation of *$result* to determine if a query was successful. After *$result* has been *#set* (added to the context), it cannot be set back to *null* (removed from the context). The details of the *#if* and *#foreach* directives are covered later in this document.

One solution to this would be to pre-set *$result* to *false*. Then if the *$query.criteria()* call fails, you can check.

```
#set( $criteria = ["spotter", "address"] )

#foreach( $criterion in $criteria )

   #set( $result = false )
   #set( $result = $query.criteria($criterion) )

   #if( $result )
      Query was successful
   #end

#end
```

Unlike some of the other Velocity directives, the *#set* directive does not have an *#end* statement.

Literals

When using the *#set* directive, string literals that are enclosed in double quote characters will be parsed and rendered, as shown:

```
#set( $directoryRoot = "data/local" )
#set( $templateName = "index.vm" )
#set( $template = "$directoryRoot/$templateName" )
$template
```

The output will be

```
 data/local/index.vm
```

However, when the string literal is enclosed in single quote characters, it will not be parsed:

```
#set( $wfo = "rfc" )
$wfo
#set( $blah = '$wfo' )
$blah
```

This renders as:

```
 rfc
 $wfo
```

By default, this feature of using single quotes to render unparsed text is available in Velocity. This default can be changed by editing `velocity.properties` such that `stringliterals.interpolate=false`.

Alternately, the *#literal* script element allows the template designer to easily use large chunks of uninterpreted content in VTL code. This can be especially useful in place of escaping multiple directives.

```
#literal()
#foreach ($woogie in $boogie)
  nothing will happen to $woogie
#end
#end
```

Renders as:

```
#foreach ($woogie in $boogie)
  nothing will happen to $woogie
#end
```

Conditionals

If / ElseIf / Else

The *#if* directive in Velocity allows for text to be included when the text product is generated, on the conditional that the if statement is true. For example:

```
#if( $test )
  THIS IS A TEST!
#end
```

The variable *$wfo* is evaluated to determine whether it is true, which will happen under one of two circumstances: (i) *$wfo* is a boolean (true/false) which has a true value, or (ii) the value is not null. Remember that the Velocity context only contains Objects, so when we say 'boolean', it will be represented as a Boolean (the class). This is true even for methods that return `boolean` - the introspection infrastructure will return a `Boolean` of the same logical value.

The content between the *#if* and the *#end* statements become the output if the evaluation is true. In this case, if *$test* is true, the output will be: "THIS A TEST!". Conversely, if *$test* has a null value, or if it is a boolean false, the statement evaluates as false, and there is no output.

An *#elseif* or *#else* element can be used with an *#if* element. Note that the Velocity Templating Engine will stop at the first expression that is found to be true. In the following example, suppose that *$wind* has a value of 15 and *$windspeed* has a value of 6.

```
#if( $wind < 10 )
   LIGHT WIND
#elseif( $wind == 10 )
   10 MPH WIND
#elseif( $windspeed == 6 )
   6 MPH WIND
#else
   GREATER THAN 10 MPH WIND
#end
```

In this example, *$wind* is greater than 10, so the first two comparisons fail. Next *$windspeed* is compared to 6, which is true, so the output is **6 MPH WIND**.

Relational and Logical Operators

Velocity uses the equivalent operator to determine the relationships between variables. Here is a simple example to illustrate how the equivalent operator is used.

```
#set ($wx = "STRONG THUNDERSTORMS")
#set ($sky = "PARTLY CLOUDY")
```

```
#if ($wfo == $rfc)

  In this case it's clear they aren't equivalent. So...
#else

  They are not equivalent and this will be the output.
#end
```

Note that the semantics of == are slightly different than Java where == can only be used to test object equality. In Velocity the equivalent operator can be used to directly compare numbers, strings, or objects. When the objects are of different classes, the string representations are obtained by calling `toString()` for each object and then compared.

Velocity has logical AND, OR and NOT operators as well. For further information, please see the VTL Reference Guide Below are examples demonstrating the use of the logical AND, OR and NOT operators.

```
## logical AND

#if( $test && $practice )
  THIS IS IN TEST AND PRACTICE MODE
#end
```

The *#if()* directive will only evaluate to true if both *$test* and *$practice* are true. If *$test* is false, the expression will evaluate to false; *$practice* will not be evaluated. If *$test* is true, the Velocity Templating Engine will then check the value of *$practice*; if *$practoce* is true, then the entire expression is true and **THIS IS IN TEST AND PRACTICE MODE** becomes the output. If *$practice* is false, then there will be no output as the entire expression is false.

Logical OR operators work the same way, except only one of the references need evaluate to true in order for the entire expression to be considered true. Consider the following example.

```
## logical OR

#if( $test || $practice )
   Test OR Practice I don't know!
#end
```

If *$test* is true, the Velocity Templating Engine has no need to look at *$practice*; whether *$practice* is true or false, the expression will be true, and **Test OR Practice I don't know!** will be output. If *$test* is false, however, *$practice* must be checked. In this case, if *$practice* is also false, the expression evaluates to false and there is no output. On the

other hand, if *$practice* is true, then the entire expression is true, and the output is **Test OR Practice I don't know!**

With logical NOT operators, there is only one argument :

```
##logical NOT

#if( !$test )
 THIS IS NOT A TEST
#end
```

Here, the if *$test* is true, then *!$test* evaluates to false, and there is no output. If *$test* is false, then *!$test* evaluates to true and **THIS IS NOT A TEST** will be output. Be careful not to confuse this with the *quiet reference $!test* which is something altogether different. There are text versions of all logical operators, including *eq*, *ne*, *and*, *or*, *not*, *gt*, *ge*, *lt*, and *le*.

One more useful note. When you wish to include text immediately following a *#else* directive you will need to use curly brackets immediately surrounding the directive to differentiate it from the following text. (Any directive can be delimited by curly brackets, although this is most useful for *#else*).

```
#if( $wfo == $rfc)it's true!#{else}it's not!#end
```

Loops

Foreach Loop

The *#foreach* element allows for looping. For example:

```
PRODUCT LIST  = :
#foreach( $product in $allProducts )
   PRODUCT : $product
#end
```

This *#foreach* loop causes the *$allProducts* list (the object) to be looped over for all of the products (targets) in the list. Each time through the loop, the value from *$allProducts* is placed into the *$product* variable.

The contents of the *$allProducts* variable is a Vector, a Hashtable or an Array. The value assigned to the *$product* variable is a Java Object and can be referenced from a variable as such. For example, if *$product* was really a Product class in Java, its name could be retrieved by referencing the *$product.Name* method (ie: *$Product.getName()*).

Lets say that *$allProducts* is a Hashtable. If you wanted to retrieve the key values for the Hashtable as well as the objects within the Hashtable, you can use code like this:

```
#foreach( $key in $allProducts.keySet() )
   Key: $key -> Value: $allProducts.get($key)
#end
```

Velocity provides an easy way to get the loop counter so that you can do something like the following:

```
#foreach( $county in $countyList )
   $velocityCount : $county.Name
#end
```

The default name for the loop counter variable reference, which is specified in the velocity.properties file, is $velocityCount. By default the counter starts at 1, but this can be set to either 0 or 1 in the `velocity.properties` file. Here's what the loop counter properties section of the `velocity.properties` file appears:

```
# Default name of the loop counter
# variable reference.
directive.foreach.counter.name = velocityCount

# Default starting value of the loop
# counter variable reference.
directive.foreach.counter.initial.value = 1
```

It's possible to set a maximum allowed number of times that a loop may be executed. By default there is no max (indicated by a value of 0 or less), but this can be set to an arbitrary number in the `velocity.properties` file. This is useful as a fail-safe.

```
# The maximum allowed number of loops.
directive.foreach.maxloops = -1
```

Include

The *#include* script element allows the template designer to import a local file, which is then inserted into the location where the *#include* directive is defined. The contents of the file **are not** rendered through the template engine. For security reasons, the file to be included may only be under TEMPLATE_ROOT.

```
#include( "one.txt" )
```

The file to which the *#include* directive refers is enclosed in quotes. If more than one file will be included, they should be separated by commas.

#include("one.txt","two.txt","three.txt")

The file being included need not be referenced by name; in fact, it is often preferable to use a variable instead of a filename. This could be useful for targeting output according to criteria determined when the template is evaluated. Here is an example showing both a filename and a variable.

#include("greetings.txt", $workfile)

Parse

The *#parse* script element allows the template designer to import a local file that contains VTL. Velocity will parse the VTL and render the template specified.

#parse("me.vm")

Like the *#include* directive, *#parse* can take a variable rather than a template. Any templates to which *#parse* refers must be included under TEMPLATE_ROOT. Unlike the *#include* directive, *#parse* will only take a single argument.

VTL templates can have *#parse* statements referring to templates that in turn have *#parse* statements. By default set to 10, the *directive.parse.max.depth* line of the `velocity.properties` allows users to customize maximum number of *#parse* referrals that can occur from a single template. (Note: If the *directive.parse.max.depth* property is absent from the `velocity.properties` file, Velocity will set this default to 10.)

Recursion is permitted, for example, if the template `dofoo.vm` contains the following lines:

```
Count down.
#set( $count = 8 )
#parse( "parsefoo.vm" )

All done with dofoo.vm!
```

It would reference the template `parsefoo.vm`, which might contain the following VTL:

```
$count
#set( $count = $count - 1 )
#if( $count > 0 )
   #parse( "parsefoo.vm" )
#else
   All done with parsefoo.vm!
#end
```

After "Count down." is displayed, Velocity passes through `parsefoo.vm`, counting down from 8. When the count reaches 0, it will display the "All done with parsefoo.vm!" message. At this point, Velocity will return to `dofoo.vm` and output the "All done with dofoo.vm!" message.

Stop

The *#stop* script element prevents any further text or references in the page from being rendered. This is useful for debugging purposes.

Evaluate

The *#evaluate* directive can be used to dynamically evaluate VTL. This allows the template to evaluate a string that is created at render time. Such a string might be used to internationalize the template or to include parts of a template from a database.

The example below will display `abc`.

```
#set($source1 = "abc")
#set($select = "1")
#set($dynamicsource = "$source$select")
#evaluate($dynamicsource)
```

Escaping VTL Directives

VTL directives can be escaped with the backslash character ("\") in a manner similar to valid VTL references.

```
## #include( "a.txt" ) renders as <contents of a.txt>
#include( "a.txt" )

## \#include( "a.txt" ) renders as #include( "a.txt" )
\#include( "a.txt" )

## \\#include ( "a.txt" ) renders as \<contents of a.txt>
\\#include ( "a.txt" )
```

Extra care should be taken when escaping VTL directives that contain multiple script elements in a single directive (such as in an if-else-end statements). Here is a typical VTL if-statement:

```
#if( $test )
   THIS IS A TEST
#end
```

If *$test* is true, the output is

THIS IS A TEST

If *$test* is false, there is no output. Escaping script elements alters the output. Consider the following case:

```
\#if( $test )
   THIS IS A TEST
\#end
```

Whether *$test* is true or false, the output will be

```
 #if($ test )
    THIS IS A TEST
 #end
```

In fact, because all script elements are escaped, *$test* is never evaluated for its boolean value. Suppose backslashes precede script elements that are legitimately escaped:

```
\\#if( $test )
  THIS IS A TEST
\\#end
```

In this case, if *$test* is true, the output is

```
\ THIS IS A TEST
\
```

To understand this, note that the `#if( arg )` when ended by a newline (return) will omit the newline from the output. Therefore, the body of the `#if()` block follows the first '\', rendered from the '\\' preceding the `#if()`. The last \ is on a different line than the text because there is a newline after 'TEST', so the final \\, preceding the `#end` is part of the body of the block.

If *$test* is false, the output is

```
\
```

Note that things start to break if script elements are not properly escaped.

```
\\\#if( $test )
   THIS IS A TEST
\\#end
```

Here the *#if* is escaped, but there is an *#end* remaining; having too many endings will cause a parsing error.

VTL: Formatting Issues

Although VTL in this user guide is often displayed with newlines and whitespaces, the VTL shown below

```
#set( $cities = ["Munetaka","Koreyasu","Hisakira","Morikune"] )
#foreach( $school in $cities )
   $school
#end
```

is equally valid as the following snippet that Geir Magnusson Jr. posted to the Velocity user mailing list to illustrate a completely unrelated point:

```
Send #set($text=["$10 and ","a pie"])#foreach($a in $text)$a#end please.
```

Velocity's behaviour is to gobble up excess whitespace. The preceding directive can be written as:

```
Send me
#set( $text = ["$10 and ","a pie"] )
#foreach( $a in $text )
$a
#end
please.
or as
Send me
#set($text       = ["$10 and ","a pie"])
            #foreach          ($a in $text )$a
        #end please.
```

In each case the output will be the same.

Other Features and Miscellany

Math

Velocity has a handful of built-in mathematical functions that can be used in templates with the *set* directive. The following equations are examples of addition, subtraction, multiplication and division, respectively:

```
#set( $total = $rain + 3 )
#set( $total = $rain - 4 )
```

```
#set( $total = $snow * 6 )
#set( $total = $sleet / 2 )
```

When a division operation is performed between two integers, the result will be an integer. Any remainder can be obtained by using the modulus (%) operator.

```
#set( $total = $snow % 5 )
```

Range Operator

The range operator can be used in conjunction with *#set* and *#foreach* statements. Useful for its ability to produce an object array containing integers, the range operator has the following construction:

```
[n..m]
```

Both *n* and *m* must either be or produce integers. Whether *m* is greater than or less than *n* will not matter; in this case the range will simply count down. Examples showing the use of the range operator as provided below:

First example:

```
#foreach( $amount in [1..5] )
$amount
#end
```

Second example:

```
#foreach( $temp in [2..-2] )
$temp
#end
```

Third example:

```
#set( $arr = [0..1] )
#foreach( $i in $arr )
$i
#end
```

Fourth example:

```
[1..3]
```

Produces the following output:
First example:
1 2 3 4 5

Second example:

2 1 0 -1 -2

Third example:

0 1

Fourth example:

[1..3]

Note that the range operator only produces the array when used in conjunction with *#set* and *#foreach* directives, as demonstrated in the fourth example.
Web page designers concerned with making tables a standard size, but where some will not have enough data to fill the table, will find the range operator particularly useful.

Advanced Issues: Escaping and !

When a reference is silenced with the *!* character and the *!* character preceded by an \ escape character, the reference is handled in a special way. Note the differences between regular escaping, and the special case where \ precedes *!* follows it:
#set($wfo = "rfc")

$\!wfo
$\!{wfo}
$\\!wfo
$\\\!wfo

This renders as:

$!wfo
$!{wfo}
$\!wfo
$\\!wfo

Contrast this with regular escaping, where \ precedes *$*:

\$wfo
\$!wfo
\$!{wfo}
\\$!{wfo}

This renders as:

$wfo

$!wfo
$!{wfo}
\rfc

String Concatenation

A common question that developers ask is How do I do String concatenation? Is there any analogue to the '+' operator in Java?.

To do concatenation of references in VTL, you just have to 'put them together'. The context of where you want to put them together does matter, so we will illustrate with some examples.

In the regular 'schmoo' of a template (when you are mixing it in with regular content) :

```
 #set( $size = "Big" )
 #set( $name = "Hail" )

The danger is $size$name.
```

and the output will render as 'The clock is BigHail'. For more interesting cases, such as when you want to concatenate strings to pass to a method, or to set a new reference, just do

```
#set( $size = "Big" )
#set( $name = "Hail" )

#set($danger = "$size$name" )

The danger is $danger.
```

Which will result in the same output. As a final example, when you want to mix in 'static' strings with your references, you may need to use 'formal references' :

```
#set( $size = "Big" )
#set( $name = "Hail" )

#set($danger = "${size}HUGE$name" )

The danger is $danger.
```

Now the output is 'The clock is BigHUGEHail'. The formal notation is needed so the parser knows you mean to use the reference '$size' versus '$sizeHUGE' which it would if the '{}' weren't there.

The Tornado Template Example

Following is the tornado template broken down into sections with descriptions to provide a better understanding of the templates.

Header

```
ZCZC ${tornadoId} DEF
TTAA00 ${wmoValue} ${dateUtil.format(${now}, ${timeFormat.ddhhmm})}
/O.NEW.${vtecOffice}.TO.W.0001.${dateUtil.format(${start},
${timeFormat.ymdthmz})}-${dateUtil.format(${expire},
${timeFormat.ymdthmz})}/
```

Headline

```
BULLETIN - EAS ACTIVATION REQUESTED
#if(${mode}=="test" || ${mode}=="practice")
TEST...TORNADO WARNING...TEST
#else
TORNADO WARNING
#end
<!-- NATIONAL WEATHER SERVICE ${officeShort} -->
${officeLong}
${dateUtil.formatLocal(${now}, ${timeFormat.header})}

#if(${mode}=="test" || ${mode}=="practice")
#set($testMessage = "THIS IS A TEST MESSAGE. ")
...THIS MESSAGE IS FOR TEST PURPOSES ONLY...
#else
#set($testMessage = "")
#end
```

Locations

```
${officeLong} HAS ISSUED A

* ${testMessage}TORNADO WARNING FOR...
#foreach (${area} in ${areas})
  ##
#if(${area.partOfArea})
#areaFormat(${area.partOfArea} 0) ##
#end
${area.name} ${area.areaNotation} IN
#areaFormat(${area.partOfParentRegion} 0) ${area.parentRegion}...
#if(${list.size($area.points)} > 0)
#if(${list.size($area.points)} > 1)
```

```
     THIS INCLUDES THE CITIES OF... #foreach (${city} in
${area.points})${city}... #end

#else
     THIS INCLUDES THE CITY OF ${list.get(${area.points},0)}
#end
#end
#end
```

Set some text values

#set($reportType1 = "TORNADO")
#set($reportType2 = "THIS TORNADO WAS")
#if(${stormType} == "line")
#set($reportType1 = "LINE OF TORNADO PRODUCING STORMS ")
#set($reportType2 = "THESE TORNADO PRODUCING STORMS WERE")
#end

Expiration time

```
* UNTIL ${dateUtil.formatLocal(${expire}, ${timeFormat.clock}, 15)}
```

Bullets

```
#set ($report = "")
#if(${list.contains($bullets, "doppler")})
  #if(${stormType} == "line")
    #set ($report = "NATIONAL WEATHER SERVICE DOPPLER RADAR INDICATED A
LINE OF SEVERE THUNDERSTORMS CAPABLE OF PRODUCING A TORNADO")
  #else
    #set ($report = "NATIONAL WEATHER SERVICE DOPPLER RADAR INDICATED A
SEVERE THUNDERSTORM CAPABLE OF PRODUCING A TORNADO")
  #end
#end
#if(${list.contains($bullets, "confirmedDoppler")})
  #set ($report = "NATIONAL WEATHER SERVICE DOPPLER RADAR WAS TRACKING
A TORNADO")
#end
#if(${list.contains($bullets, "confirmedLarge")})
  #set ($report = "NATIONAL WEATHER SERVICE DOPPLER RADAR WAS TRACKING
A LARGE AND EXTREMELY DANGEROUS TORNADO")
#end
#if(${list.contains($bullets, "spotter")})
  #set ($report = "TRAINED WEATHER SPOTTERS REPORTED A TORNADO")
#end
#if(${list.contains($bullets, "lawEnforcement")})
  #set ($report = "LOCAL LAW ENFORCEMENT REPORTED A TORNADO")
#end
#if(${list.contains($bullets, "public")})
  #set ($report = "THE PUBLIC REPORTED A TORNADO")
#end
#if(${list.contains($bullets, "spotterFunnelCloud")})
```

```
  #set ($report = "TRAINED WEATHER SPOTTERS REPORTED A FUNNEL CLOUD")
#end
```

Path Cast

```
#set($closest = ${list.get($closestPoints, 0)})
#set($secondary = ${list.get($closestPoints, 1)})
* ${testMessage}AT ${dateUtil.formatLocal(${event},
${timeFormat.clock})}...${report} ##
#if($closest.roundedDistance <= 4)
OVER ##
#else
${closest.roundedDistance} MILES #direction(${closest.roundedAzimuth})
OF ##
#end
${closest.name}...OR ${secondary.roundedDistance} MILES
#direction(${secondary.roundedAzimuth}) OF ${secondary.name}##
#if($movementInMph < 2.5)

${reportType2} NEARLY STATIONARY.
#else
...MOVING #direction(${movementDirectionRounded}) AT
${mathUtil.round(${movementInMph})} MPH.
#end

## Determine if the pathcast goes over only rural areas
#set($ruralOnly = 1)
#foreach(${pc} in ${pathCast})
#if(${pc.points})
#set($ruralOnly = 0)
#end
#end
#set($numOtherCities = ${list.size($otherPoints)})

#if(${pathCast} && ${ruralOnly} == 0)
#if(${stormType} == "line")
* ${testMessage}THE TORNADO PRODUCING STORMS WILL BE NEAR...
#else
* ${testMessage}THE TORNADO WILL BE NEAR...
#end
#set($numRural = 0)## indicates the number of rural points
#foreach (${pc} in ${pathCast})
#if(${pc.points})
#set($numRural = 0)
#set($numCities = ${list.size($pc.points)})
#set($count = 0)
  ##
#foreach (${city} in ${pc.points})
#if(${city.roundedDistance} < 3)
## close enough to not need azran, considered OVER the area
${city.name}##
#else
## needs azran information
${city.roundedDistance} MILES #direction(${city.roundedAzimuth}) OF
${city.name}##
```

```
#end
#set($count = $count + 1)
#if($count == $numCities - 1)
 AND ##
#elseif($count < $numCities)
...##
#end
#end
 BY ${dateUtil.formatLocal(${pc.time}, ${timeFormat.clock})}...
#else## Handle the rural cases
#set($numRural = $numRural + 1)
#if($numRural > 2)
  RURAL ${pc.area} ${pc.areaNotation} AT
${dateUtil.formatLocal(${pc.time}, ${timeFormat.clock})}...
#set($numRural = 0)
#end
#end
#end
#elseif(${otherPoints} && ${numOtherCities} > 0)
* ${testMessage}OTHER LOCATIONS IN THE WARNING INCLUDE BUT ARE NOT
LIMITED TO
#set($count = 0)
  ##
#foreach(${point} in ${otherPoints})
#set($count = $count + 1)
${point}##
#if($count == $numOtherCities - 1)
 AND ##
#elseif($count < $numOtherCities)
...##
#end
#end
#else
* ${testMessage}THE ${reportType1} WILL OTHERWISE REMAIN OVER MAINLY
RURAL AREAS OF THE INDICATED COUNTIES.
#end
```

Call to Action Statements

```
#if(${list.contains($bullets, "hailWinds")})
${testMessage}IN ADDITION TO THE TORNADO...THIS STORM IS CAPABLE OF
PRODUCING !**EDIT SIZE**! SIZE HAIL AND DESTRUCTIVE STRAIGHT LINE
WINDS.

#end
#if(${list.contains($bullets, "severeDoppler")})
${testMessage}WHEN A TORNADO WARNING IS ISSUED BASED ON DOPPLER
RADAR...IT MEANS THAT STRONG ROTATION HAS BEEN DETECTED IN THE STORM.
A TORNADO MAY ALREADY BE ON THE GROUND...OR IS EXPECTED TO DEVELOP
SHORTLY.  IF YOU ARE IN THE PATH OF THIS DANGEROUS STORM...MOVE INDOORS
AND TO THE LOWEST LEVEL OF THE BUILDING.  STAY AWAY FROM WINDOWS.  IF
DRIVING...DO NOT SEEK SHELTER UNDER A HIGHWAY OVERPASS.

#end
#if(${list.contains($bullets, "severe")})
```

```
${testMessage}THIS IS AN EXTREMELY DANGEROUS AND LIFE THREATENING
SITUATION. THIS STORM IS CAPABLE OF PRODUCING STRONG TO VIOLENT
TORNADOES. IF YOU ARE IN THE PATH OF THIS TORNADO...TAKE COVER
IMMEDIATELY!

#end
#if(${list.contains($bullets, "overpasses")})
${testMessage}DO NOT USE HIGHWAY OVERPASSES FOR SHELTER.  OVERPASSES DO
NOT PROVIDE PROTECTION FROM TORNADIC WINDS.  VEHICLES STOPPED UNDER
BRIDGES BLOCK TRAFFIC AND PREVENT PEOPLE FROM GETTING OUT OF THE
STORM'S PATH AND TO SHELTER.  IF YOU CANNOT DRIVE AWAY FROM THE
TORNADO...GET OUT OF YOUR VEHICLE AND LIE FLAT IN A DITCH AS A LAST
RESORT.

#end
#if(${list.contains($bullets, "outside")})
${testMessage}IF YOU ARE CAUGHT OUTSIDE...SEEK SHELTER IN A NEARBY
REINFORCED BUILDING.  AS A LAST RESORT...SEEK SHELTER IN A
CULVERT...DITCH OR LOW SPOT AND COVER YOUR HEAD WITH YOUR HANDS.

#end
#if(${list.contains($bullets, "outrun")})
${testMessage}DO NOT USE YOUR CAR TO TRY TO OUTRUN A TORNADO.  CARS ARE
EASILY TOSSED AROUND BY TORNADO WINDS.  IF YOU ARE CAUGHT IN THE PATH
OF A TORNADO...LEAVE THE CAR AND GO TO A STRONG BUILDING.  IF NO SAFE
STRUCTURE IS NEARBY...SEEK SHELTER IN A DITCH OR LOW SPOT AND COVER
YOUR HEAD.

#end
#if(${list.contains($bullets, "mobileHome")})
${testMessage}IF IN MOBILE HOMES OR VEHICLES...EVACUATE THEM AND GET
INSIDE A STURDY SHELTER.  IF NO SHELTER IS AVAILABLE...  LIE FLAT IN
THE NEAREST DITCH OR OTHER LOW SPOT AND COVER YOUR HEAD WITH YOUR
HANDS.

#end
#if(${list.contains($bullets, "rainWrapped")})
${testMessage}HEAVY RAINFALL MAY OBSCURE THIS TORNADO. TAKE COVER NOW!
IF YOU WAIT TO SEE OR HEAR IT COMING...IT MAY BE TOO LATE TO GET TO A
SAFE PLACE.
#end
#if(${list.contains($bullets, "enterReport")})
!** Enter Any Storm Reports Here **!
#end
#if(${list.contains($bullets, "safePlaces")})
${testMessage}THE SAFEST PLACE TO BE DURING A TORNADO IS IN A BASEMENT.
GET UNDER A WORKBENCH OR OTHER PIECE OF STURDY FURNITURE.  IF NO
BASEMENT IS AVAILABLE...SEEK SHELTER ON THE LOWEST FLOOR OF THE
BUILDING IN AN INTERIOR HALLWAY OR ROOM SUCH AS A CLOSET.  USE BLANKETS
OR PILLOWS TO COVER YOUR BODY AND ALWAYS STAY AWAY FROM WINDOWS.

IF IN MOBILE HOMES OR VEHICLES...EVACUATE THEM AND GET INSIDE A
SUBSTANTIAL SHELTER.  IF NO SHELTER IS AVAILABLE...  LIE FLAT IN THE
NEAREST DITCH OR OTHER LOW SPOT AND COVER YOUR HEAD WITH YOUR HANDS.
#end

#if(${mode}=="test" || ${mode}=="practice")
```

```
THIS IS A TEST MESSAGE. DO NOT TAKE ACTION BASED ON THIS MESSAGE.
#end
```

TML lines

```
LAT...LON ##
#foreach(${coord} in ${areaPoly})
#llFormat(${coord.y}) #llFormat(${coord.x}) ##
#end

TIME...MOT...LOC ##
${dateUtil.format(${event}, ${timeFormat.time})}Z ##
${mathUtil.round(${movementDirection})}DEG ##
${mathUtil.round(${movementInKnots})}KT ##
#llFormat(${eventLocation.y}) #llFormat(${eventLocation.x})

$$
!**NAME/INITIALS**!
```

REFFERENCE: The Apache Software Foundation Last Published: 2007-05-02 07:39:31